

SecurityLib 1.0

by
MarVic

Interface and programming guide

Table of contents:

INTRODUCTION.....	2
Who should read it?	2
General information	2
When TSecRights object is const?	2
__stdcall vs. __fastcall	3
SYSTEM INTERFACE.....	3
What is it?	3
Security manipulating routines	3
Auto login password protection.....	5
System data common to whole application	5
SECURITY INTERFACE.....	5
Logging in Security object	6
Logging out	7
Showing administrator dialog.....	7
Showing user dialog	7
Modifying user rights and parameters programmatically	7
OBJECT INTERFACE	8
Constructors	8
User interface	8
Miscellaneous functions	9
RIGHTS AND PARAMETERS INTERFACE.....	9
Rights functions.....	9
Parameters functions	10
ADDITIONAL TYPES, CLASSES AND FUNCTIONS.....	11
PARAMETERS EDITOR INTERFACE.....	11
Base and child parameters.....	12
Exports	12
Autonames feature.....	12
Key security feature.....	12

TSecSmpParamEd and TSecParamEd interface	13
Using not child parameters and database parameters	13
DATABASE PARAMETERS EDITOR.....	13
PASSWORDS GUIDELINES	13
SecurityLib database passwords.....	13
SecurityLib's passwords	14
SECUREDFORM UNIT.....	14
DB PROVIDERS.....	14
What is it?	14
Registry	14
Enclosed DB providers	15
Documents.....	15
OTHER	15
Other documents:	15

Introduction

Who should read it?

Well, programmers who wish to use the library.

General information

Whole functionality of library is encapsulated in two classes: TSecRights and TSecParamEd.. The first is used to perform standard tasks, as connecting to SecurityLib database or checking password, while the latter contains functions useful during the execution of custom property editor. The header files for those classes are, respectively, Rights.h (Rights.pas, Rights.bas) and SecParamEd.hpp (SecParamEd.pas).

The implementation of SecurityLib uses VCL (Borland's Delphi/C++Builder Visual Component Library) dlls. The names depends on the version of the SecurityLib and VCL, which has to match. Both SecurityLib dll file and VCL files names (VCLx.bpl and VCLDBx.bpl, x stands for 4 or 5 or later) end with number determining the version.. You have to distribute them with your application. It is strongly recommended that if your application uses VCL too, SecurityLib should use the same version as the program. If not, you will unnecessarily enlarge your application size by shipping two copies of VCL. If the correct version of SecurityLib is not available, contact me.

When using SecurityLib in C++ applications, you should add SecurityLibx.lib to your project to perform static linking of dll functions. In other programming languages, the link is done dynamically.

The TSecRight objects is used as a connection between user and an object called Security, which takes care of all tasks except for administrator windows.. This object is hidden (not accessible for user). Many TSecRights objects can point to one Security object - so you can access your rights and parameters from many places. In fact, TSecRights functions like a pointer, as it simply contains one.

When TSecRights object is const?

And because of that, when TSecRight is const, it means that Security object should remain unchanged. So, for example, retrieving parameters, rights or reading properties like Logged can be done on const object, but doing all other things modifies the Security object, so they are allowed only on non-const TSecRight objects.

__stdcall vs. __fastcall

There are two versions of TSecRights class. Methods of original one use Borland's fastcall calling convention, while the TSecStdRights use stdcall convention for compatibility with other programming languages. The definition of second class is stored in StdRights.h file.

System interface

What is it?

The system interface is part of TSecRights methods, which are static, and are used to manipulate objects that are common to whole library, for example objects for accessing SecurityLib databases. Here we have two major groups of routines.

Security manipulating routines

Introduction

These methods are used to add, delete and find objects called Security. Each Security object corresponds to one SecurityLib database connection (in the future, Security objects may be represented as TSecurity Delphi/C++Builder design and runtime components). Database can be accessed only through these objects. They are created dynamically, and the list of currently allocated Securities is maintained. Then, if you want to get rights and params information for user, you have to know the name of Security object, which corresponds to your database.

Adding Securities

There are four methods in TSecRights class used to add Securities. Each of them will set different properties of created object.

When you create Security, you have to choose type of so-called auto login. You choose from following possibilities:

```
enum SecAutoLoginType {
    altNone, // no auto logging, have to log manually
    altFixedPasswd, // fixed password given at creation of Security
    altSimplePasswd, // shows passwd dialog and asks only once for
password
    altPasswdDlg, // shows passwd dialog and ask about password till
it's entered correctly or users exits dialog without passwd
    altCustom // allows to write custom login event handler
};
```

First seven parameters and last two ones is common to all of methods:

```
static TSecurity * __fastcall AddSecurity(const char *secname,
const char *alias, const char *dbprov, const char *dbend, void * data,
const char *secuser, const char *secencpswd, /*... */ SecEncodeFunc
f=0, bool autolog = true);
```

Secname is the unique ID of this Security, and adding two objects with same name may result in unexpected behavior of library. Alias should point to security database, but the form of it depends on the type of [DB Provider](#) used. In case of BDE it is a BDE alias, in case ADO it is connection string.

DBProv it is a string which contains the name of DB Provider, as set in the registry. These names can be found in documentation of particular providers. DBEng is a name of DB Engine to use by provider. Sometimes one provider – dll file may expose many types of databases – they can be selected here. For names refer to documentation of providers. Data is a void pointer that will be passed to provider. Different providers can accept different values.

Secuser and secencpasswd are User and Password parameters for logging into SecurityLib database. Password may be encrypted, as it's stored in memory whole time Security object exists, and could possibly be stolen. However, the read-only password, which you are supposed to pass, typically is public ([see Passwords Guidelines](#)). Decryption routine is called every time password is needed. Default password decryption routine just does nothing – that is returns the same string it got. More information about decryption procedures in section [Auto Login Password Protection](#) later this chapter.

The second last parameter is quite obsolete – it's pointer to SecurityLib user passwords encryption function. Now it is advised to use standard encryption functions, because it is secure enough, and it lets other programs to log into your SecurityLib database.

Autolog parameter indicates if Security object should maintain a counter of currently binded to it TSecRights objects. If it's true, binding the first TSecRights object to this Security object, will invoke AutoLogin function (only if it wasn't logged in already). Destroying (or unbinding) the last TSecRights bound to that Security object will log out from security database.

All functions return pointer to TSecurity – which encapsulates Security object. You can't manipulate TSecurity objects directly – there is no definition for this class. Only possible way to affect Security object is to use static methods of TSecRights.

And now each method is described:

```
AddSecurity(const char *secname, const char *alias, const char
*dbprov, const char *dbend, void * data, const char *secuser, const
char *secencpasswd, SecEncodeFnc f=0, bool autolog = true);
```

This one sets altNone auto login mode.

```
AddSecurity(const char *secname, const char *alias, const char
*dbprov, const char *dbend, void * data, const char *secuser, const
char *secencpasswd, const char *fullname, SecEncodeFnc f=0, bool autolog =
true);
```

This one sets altSimplePasswd login type. Fullname parameter will be displayed in password dialog as Security database name (in order to let user know which password should he use).

```
AddSecurity(const char *secname, const char *alias, const char
*dbprov, const char *dbend, void * data, const char *secuser, const
char *secencpasswd, const char *user, const char *passwd, SecEncodeFnc f=0,
bool autolog = true);
```

This one sets altFixedPasswd login mode. Parameters user and password are used in logging in SecurityLib.

```
AddSecurity(const char *secname, const char *alias, const char
*dbprov, const char *dbend, void * data, const char *secuser, const
char *secencpasswd, const char *error, const char *wrongpasswd, const char
*t, const char *fullname, SecEncodeFnc f=0, bool autolog = true);
```

This one sets altPasswdDialog mode. Given error message is displayed, when connection to SecurityLib database could not be established. Wrongpasswd message is displayed, when password entered in dialog was incorrect, and t is the title of that message.

```
AddSecurity(const char *secname, const char *alias, const char
*dbprov, const char *dbend, void * data, const char *secuser, const
char *secencpasswd, SecOnLogin onlogin, void *autodata, bool autolog =
true);
```

This one sets altCustom mode. SecOnLogin type is defined as follows

```
typedef bool __stdcall (*SecOnLogin) (TSecurity *co, void *);
```

Routine gets TSecurity pointer it is supposed to log on, and void pointer, which was parameter of AddSecurity function and is intended for custom programmer use. It should return true if and only if logging was successful

You can log in by calling AutoLogin method of TSecRights.

Obtaining Security by name

There is special routine, which simply gets Security object name as parameter and returns TSecurity pointer:

```
static TSecurity * __fastcall FindSecurity(char *name);
```

Removing Security object

```
static TSecurity * __fastcall DeleteSecurity(char *name);
```

I think it's clear what it does and how to use it.

Auto login password protection

Because SecurityLib database password given while creation of Security object is stored as its property, it has to be encrypted to provide maximum security. Password will be decoded each time user tries to log in.

```
typedef char * __stdcall (*SecDecodeFnc)(const char *enc, char  
*dec);
```

Function gets encrypted password and buffer for decrypted password as a parameters. After decoding password, you should return it.

```
static void __fastcall SetPasswdDecoder (SecDecodeFnc co );
```

Above function sets decrypts function for all Security objects. It's common for all objects, because it real password is independent of it's encrypted form, and you can use one function for decrypting all SecurityLib database's passwords.

This is older part of code, SecurityLib's database read-only password no longer has to be secret (unless you want not-encrypted SecurityLib data to be secret) because only SecurityLib knows how to obtain read-write password.

System data common to whole application

In this group there are functions, which sets properties influencing whole SecurityLib.

```
static void __fastcall SetAppHwnd(HWND app);
```

Call it as soon as the application main window handle is available. This is used to integrate SecurityLib modal windows and message boxes with your application.

```
static void __fastcall SetMDIClient(HWND app);
```

This is thought to set the MDI client area of the main window, when you want the Security Administrator and other windows to behave as MDI child windows. Currently not available.

```
static HWND __fastcall GetMDIClient();
```

Returns the handle of MDI client area.

Security interface

Here are routines which operate on Security objects created using functions described in previous chapter.

There are two groups of functions in this chapter. One group contains only static methods and needs Security object to be explicitly specified while the second uses Security object attached to TSecRights object.

Logging in Security object

This is very first thing you do after adding a Security (if you did not turned on the Autolog function). You can choose between manual and automatic logging.

Automatic logging

If all needed data are stored in Security object, all you have to do is call AutoLogin function.

```
bool __fastcall AutoLogin();
static bool __fastcall AutoLogin(TSecurity *);
```

You can call this function before or after showing your main window or when you need to access the rights.

Manual logging

This case is much more complicated, because you have to remember about preparing and unpreparing logging.

```
static bool __fastcall PrepareLogin(TSecurity *s);
bool __fastcall PrepareLogin();
```

Above function prepares Security object for logging in, including connecting to database, checking its signature and allocating some memory. It **has** to be called before any login function. Returns true on success and false on failure.

```
static bool __fastcall PrepareLoginWindow(TSecurity *s,char
*fullname);
bool __fastcall PrepareLoginWindow(char *fullname);
```

This one prepares standard password dialog box. It calls also PrepareLogin. Fullname will be displayed in dialog as Security database name.

```
bool __fastcall Login(SecAutoLoginType t);
static bool __fastcall Login(TSecurity *, SecAutoLoginType t);
```

It forces autologin of specified type. Currently useless, reserved for future use. Returns true if login was successful.

```
bool __fastcall Login();
static bool __fastcall Login(TSecurity *);
```

Displays password dialog. Returns true if OK button was clicked, otherwise false.

```
bool __fastcall Login(const char *name,const char *passwd);
static bool __fastcall Login(TSecurity *, const char *name,const
char *passwd);
```

Tries to log in with specified user name and password to Security object. Returns true on success.

```
bool __fastcall Login(const char *error,const char *wrongpasswd,
const char *t, const char *fullname);
static bool __fastcall Login(TSecurity *, const char *error,const
char *wrongpasswd, const char *t, const char *fullname);
```

Displays password dialog until user enters correct user name and password or clicks cancel. If password was correct, function logs in. Returns true if logged into Security object. Parameters have same meaning as in [AddSecurity](#) function.

NOTE: This functions executes the PrepareLoginWindow().at the beginning and PostLogin before exiting, so you don't have to care about it.

```
static void __fastcall PostLogin(TSecurity *s);
void __fastcall PostLogin();
```

This should be called after unsuccessful login; technically after succesful call to PrepareLogin or PrepareLoginWindow. It releases memory allocated for logging in. When the user has succefully logged in, you don't have to call this function (it's called internally), but it is allowed and has no negative consequences (including time penalties).

Logging out

It's much simpler. Just call following routine:

```
void __fastcall Logout();
static void __fastcall Logout(Tsecurity *);
```

If you are not sure if user is logged into Security object, call

```
bool __fastcall Logged();
static bool __fastcall Logged(Tsecurity *);
```

which returns true if someone is logged in this object.

Showing administrator dialog

```
void __fastcall ShowDlg(HWND app, SecDBEditEncodeFnc fnc = 0);
static void __fastcall ShowDlg(TSecurity *co, HWND app, bool
mdichild, SecDBEditEncodeFnc fnc = 0);
```

It shows modal Security Administrator dialog. App parameter is handle of parent window – it is used to integrate with your application. Mdichild parameter determines if dialog should be showed as MDI interface child window. Fnc is used for encrypting password of *Database Parameters Editor*, which is dialog box provided for you to easily store user name and password for your program database (when it uses one).

```
typedef SecString __stdcall (*SecDBEditEncodeFnc)(const char *co,
int paramkey, int userkey);
```

It gets string to encrypt, and two keys, which should be used to encrypt data (that is, output string should vary on that values) to ensure maximum security of your password. It should return encrypted password.

Showing user dialog

```
void __fastcall ShowSmpDlg(SecDBEditEncodeFnc fnc = 0, bool
mdichild =false);
static void __fastcall ShowSmpDlg(TSecurity *, SecDBEditEncodeFnc
fnc = 0, bool mdichild =false);
```

This function is similar to the previous one, except it shows user dialog, which enables to view rights and modify parameters only of currently logged user.

Modifying user rights and parameters programmatically

SecurityLib enables you to manipulate users programmatically, for example to create your own user manager. This is done using the parameters editor interface.

```
TSecParamEd *__fastcall ModifyStart(const PswdString &key);
void __fastcall ModifyEnd(TSecParamEd *co);
```

Calling ModifyStart will open Security database in write mode, and return a pointer to editor interface. More info on using the interface in the chapter Parameters editor interface.

The use of this function is restricted. You have to know the key to make use of this function. If you want to use it, [contact me](#), and I will give you the key.

Object interface

In this part there are functions which operate on Security objects, but have no static versions. There are though to be used with TSecRights object, which stores the reference to Security object. As they are small and there are many of them, I think it would be not comfortable to use them passing TSecurity * as a parameter. So if you are going to make many operations on Security object, you should create TSecRights variable.

Constructors

```
__fastcall TSecRights() {}
```

Default constructor. Does nothing.

```
__fastcall TSecRights(TSecurity *co);
```

Creates TSecRights object and binds it to the given Security object. If the Autolog mode for Security is on and the Security was not logged into Security database, it tries to log in. In case of failure, throws [ESecAutoLoginFailed](#) exception.

```
__fastcall TSecRights(const char *name);
```

Creates TSecRights object and binds it to the Security object which name was given. If it was not found, [Binded](#) function returns false. If the Autolog mode for Security is on and the Security was not logged into Security database, it tries to log in. In case of failure, throws [ESecAutoLoginFailed](#) exception.

```
__fastcall TSecRights(const TSecRights &co);
```

Copy constructor. Binds new TSecRights object to the same Security the given one was. Also active right is copied.

User interface

```
SecString __fastcall UserName() const;
```

Returns the name of logged user in the Security object pointed by this TSecRights object.

```
FullString __fastcall UserFullName() const;
```

Returns the full name of logged user in the Security object pointed by this TSecRights object.

```
PswdString __fastcall UserPasswd() const;
```

Returns the encrypted password of logged user in the Security object pointed by this TSecRights object.

```
int __fastcall UserID() const;
```

Returns the unique ID of logged user in the Security object pointed by this TSecRights object.

```
int __fastcall GroupID() const;
```

Returns the unique ID of logged user group in the Security object pointed by this TSecRights object.

```
SecPasswdOption __fastcall UserAllowChangePswd() const;
```

```
SecPasswdOption __fastcall UserAllowNullPasswd() const;
```

```
SecPasswdOption __fastcall UserNoChPswdAtLogin() const;
```

```
SecPasswdOption __fastcall UserAccountEnabled() const;
```

```
SecPasswdOption __fastcall UserPasswdControl() const;
```

These five functions return each password option. The possible return values are:

```
enum SecPasswdOption {
```

```

    poFalse,    // user has that right
    poTrue,     // user has not that right
    poGroup     // right of user is the same as group
};

```

The password option names are changed to reflect the return value: sometimes it's a negation of option showed in the change password box. This is because I wanted those values to be true only if it gives some right or privilege to the user. In the following table there are the same options with both names:

Function name	Negated	Name
PasswdControl	No	Password Control
NoChPswdAtLogin	Yes	Change password at login
AllowNullPasswd	Yes	Disallow null password
AllowChangePswd	No	Allow changing password
AccountEnabled	Yes	Account disabled

```

bool __fastcall GroupAllowChangePswd() const;
bool __fastcall GroupAllowNullPasswd() const;
bool __fastcall GroupNoChPswdAtLogin() const;
bool __fastcall GroupAccountEnabled () const;
bool __fastcall GroupPasswdControl () const;

```

These five functions return the group password options. They're returning bool, as option for groups can have only two states: granted or denied.

```

bool __fastcall GroupForceAddMod() const;

```

This function returns the [force add to above groups right](#). When is true, when adding new group all groups that are above the creator's group are set to be above the new group.

```

bool __fastcall GetAllowChangePswd() const;
bool __fastcall GetAllowNullPasswd() const;
bool __fastcall GetNoChPswdAtLogin() const;
bool __fastcall GetAccountEnabled() const;
bool __fastcall GetPasswdControl() const;

```

Those functions combine previous User... and Group... ones. They return effective state of the option: if the value of user option is poGroup, they return Group... function value; else they just return User... function value.

Miscellaneous functions

```

bool __fastcall Binded() const;

```

Returns true if TSecRights object has any Security object associated.

Rights and parameters interface

Functions in this interface enable you to work with rights and parameters values. You will use those functions when you want to obtain those values i.e. to protect your windows or print a form. Parameters are maximum 16 chars long.

Rights functions

```

int __fastcall GetRight(const char *name) const;

```

Function retrieves right, which name matches parameter, from Security database, and puts its value to the Rights data member. Returns unique ID of right. If there is no right with such a name function returns -1 and set Rights to 0 – all basic rights not granted.

```
int __fastcall GetRight(int FormID) const;
```

Function returns right with identification number FormID. Identification number may be obtained from previous calls to GetRight(const char *).

This function was thought to eliminate unnecessary searches against list of rights, but there won't be big penalty of speed, if you call the name version.

```
enum SecRightsMask { rmView=1, rmInsert=2, rmModify=4, rmDelete=8
};
union {
    mutable SecRightsMask Rights;
    struct {
        mutable bool View :1;
        mutable bool Insert:1;
        mutable bool Modify:1;
        mutable bool Delete:1;
    };
};
```

This public data member is used to read right first retrieved by GetRight functions. You can use Rights member to pass rights as parameters or perform bit wise operations. Reference View, Insert, Modify and Delete members to get the appropriate basic right value.

```
operator SecRightsMask&() const { return Rights; }
```

This self-explanatory operator returns the Rights data member value.

Parameters functions

```
SecString __fastcall GetParam(const char *name) const;
SecString __fastcall GetParam(int ParamID) const;
```

Those two functions return a value of parameter as SecString. The parameter may be passed by name or by ID, which currently cannot be obtain from any function. If there is no parameter with specified name/ID, the functions return empty string. Can be used for any type of parameter.

```
PswdString __fastcall GetPswdParam(const char *name) const;
PswdString __fastcall GetPswdParam(int ParamID) const;
```

Functions return a value of specified parameter as a PswdString. There is no processing done during the process of writing data to the database, particularly no transliterations, and it is meant to operate on encrypted strings. Returns empty string when no parameter could be found.

NOTE: some database systems do not store data after the terminating NULL. It is not guaranteed, that chars after NULL will be the same as those that were stored. It depends on your DB Provider.

```
bool __fastcall GetBoolParam(const char *name) const;
bool __fastcall GetBoolParam(int ParamID) const;
```

Return true only if the value of passed parameter name/ID is equal to the YesString defined in database. Can be used for any type of parameter. If functions could not find parameter, return false.

```
int __fastcall GetIntParam(const char *name) const;
int __fastcall GetIntParam(int ParamID) const;
```

Return the value of passed parameter after converting it to the integer. If the value was not an integer, or parameter could not be found, returns INT_MIN.

```
SecString __fastcall GetDBParam(const char *name) const;
bool __fastcall GetBoolDBParam(const char *name) const;
```

```
PswdString __fastcall GetPswdDBParam(const char *name) const;
```

Functions return the value of Security database parameter, which name was passed as an argument. If there was no parameter with such a name, return empty string, false, or INT_MIN.

NOTE: some database systems do not store data after the terminating NULL. It is not guaranteed, that chars after NULL will be the same as those that were stored. This is handled in SecurityLib functions, which need passwords.

Additional types, classes and functions

```
class ESecExcept: public exception {};
```

Defines base exception for all exceptions thrown by library.

Unfortunately, library cannot guarantee all exceptions thrown while executing library functions will derive from this class. As library uses VCL, and those functions can also throw its own exceptions, embracing all functions in try .. catch block to converting exceptions to another ones would be very expensive. Therefore I decided not to do this. I think those exceptions will be thrown only in few cases, i.e. when someone corrupted database file during execution of program, and you can catch them anyway (esp. when using VCL).

```
class EDBProviderError: public ESecExcept {};
```

This exception class is thrown when there was a database provider error.

```
class EDBProviderDModError: public EDBProviderError {};
```

The exception is thrown when there was an error connected with database provider.

```
class EsecAutoLoginFailed: public EsecExcept {};
```

Thrown when the AutoLogin function was called by Bind() or TSecRights constructor, and the user could not login into SecurityLib or during execution of PrepareLogin there was an error.

```
#define ParamValLen 16
```

```
#define FullNameLen 50
```

```
typedef smallstr<ParamValLen> SecString;
```

```
typedef smallstr<FullNameLen> FullString;
```

```
typedef pswdsstr<ParamValLen> PswdString;
```

This defines three string types used in library. The first one is used in all functions returning parameter values as text. The second one is longer and used only for full names, and finally, PswdString is used to operate on key and passwords, it preserves chars even after terminating null.

The smallstr<T> and pswdsstr<T> types are defined in strings.hpp header file. The implementation uses static char types (arrays with fixed size), so no new/delete operations are needed. (if SecurityLib returned the dynamically allocated string, it could not be freed in a normal way, but it would have to be returned to library for destruction).

```
class TSecurity;
```

This is declaration of TSecurity data type. This release of SecurityLib does not include the definition of the class.

Parameters editor interface

This interface is meant for custom parameter editors., that is, small dialog boxes, presenting the information from standard parameters in their own form. With SecurityLib is shipped one example of editor, called *User Data Editor* along with source code (Borland C++Builder 4/5). You can view it to learn how to write your own one.

In this chapter only the most important things will be discussed, as usually your editor will differ from example one only in the number and names of children parameters. Usually, the editors are used to present values of standard parameters in more convenient and friendly way.

Base and child parameters

Each custom parameter, when added into Parameter list, is seen by system as the parameter of Default type (although its type name is different). That means that always under the name of that custom parameter is a value, that you can get using standard functions. Also that value can be changed by user without invoking the editor. You can store additional data in child parameters (subparams) of other types. Those parameters are strictly connected to your one: when your parameter is renamed, the child ones are also. The same thing with removal.

Exports

The library should export four basic functions:

```
SecString _export __stdcall ShowUserEditor(const TSecSmpParamEd &co,
bool &cancel);
bool _export __stdcall InsertNewParam(const TSecParamEd &co);
bool _export __stdcall IterateThroughChildren(const TSecParamEd &co);
bool _export __stdcall RenameParam(const TSecParamEd &co);
```

Each of them gets as a parameter TSecParamEd interface. Using its functions, you should respond to the call.

The first function is called, when user clicks the button with three dots in Parameters Values list. It should show your custom window. After interacting with the user, it should indicate, if the user wanted to apply changes, or discard them. You should return new value for base parameter (if it is needed).

The second routine will be called when user inserts new parameter of that type. The base parameter is already inserted. You may display information window, ask for some options considering parameter settings. Return true on success.

The iterate function is used to delete all parameters or perform other task on all of child parameters. You are supposed to call a function passed on all of your parameters. Return true if all function calls returned true.

The last one should rename all subparams and return true if successful.

The function can have any names as they have to be entered in *Parameter Types* dialog. The parameter editor library is unloaded after each use of one of the functions.

Autonames feature

To ease the necessity to enter the names of functions in the *Parameter Types* dialog box, they can be filled automatically (see Admin's Manual). To do that, you have to export one more function:

```
void __stdcall _export _MarSecurityGetFunctionNames(TSecFncNames *co,
HWND app);
```

It has to have exactly that name. In this function you should set the TSecFncNames object members to appropriate values. Also you should show some information about the editor to enable user to recognize the library.

Key security feature

When using autonames, you can also pass name of so called key function and an encrypted key. This is to prevent hackers from changing the dll file of editor. The key, encrypted for the first time now, will

be saved in a once more encrypted form in database. Then, it will be always checked if the function encrypts the key in the same way.

You have to protect your encrypt method and source with that code, because if you loose it, there will be no other way that changing all copies of your editor file for one with new method.

TSecSmpParamEd and TSecParamEd interface

Most part of that interface should be clear, so I will wrote only couple of words.

There are two interfaces, because when only showing editor, there may be no capability to add or remove parameters as it can be called from *User Security* window. So, using TSecSmpParamEd you cannot access some of functions.

The ...Sub... versions of GetParam, AddParam and all other functions does the same thing what the usual routine except for it prepends the name you specified as a parameter with base parameter name. This is just the simpler way to access child parameters.

The Def... functions reverts right's or parameter's value to group's default one.

Using not child parameters and database parameters

You may want to add a parameter with a unique name, which won't be renamed, and it will be common to all of parameters of this type or even across different parameter types. You have to remember to remove it after last of your parameters was deleted.

You can also access DB parameters and change theirs value. This may be the way to communicate between parameters of different users.

Database Parameters Editor

It is built-in property editor. It is thought to be used to store user's account to log in to the main program database - in case not all users have separate account in that database. You may create accounts only for groups, and then use that parameter value to log in.

The password is stored in encrypted form as a separate parameter. You have to provide the [encrypting function](#). If you write it correctly - the output should vary on both keys - neither copying the value from other user nor entering it manually in *Show Hidden* mode will enable hacker to get unauthorized access.

Passwords Guidelines

This chapter covers the topics of various passwords in the system.

SecurityLib database passwords

You should get your SecurityLib database with DB Provider. Also, you should know how to change the passwords. In that database there should be at least three accounts created.

Read-only account

This account is used at the begging by Securty object when preparing login. The password may be public, unless you want the information about user rights and parameters to be secret. All substantial internal data is encrypted, so you don't have to worry about someone stealing full access.

Read-write account

This one is used by *User Administrator*. SecurityLib needs full access to all its tables and queries. The password is stored in SecurityLib database (of course encrypted). To set it, there is a special utility included in the package, called PasswdSetter. You should enter there connection data (as a database account use Administrator's account) and password. Using this program you can also register the database. This password is not public.

Administrator's account

There should be one with full access, and possibility to change other's account passwords. When you get the database with DB Provider, it should be empty, but you have to change it before distribution. You may want to give away the password to your customers - sometimes (for example if SecurityLib's Admin forgets his password) there is no other way than manually modifying the passwords of users - you can it to be empty. This password is also strictly confidential. It is the best if each customer has different passwords.

SecurityLib's passwords

These are stored in database in encrypted form. You should not modify it without using SecurityLib. The only reason may be forgotten password. In that case you should delete all chars from Password field of Users table.

SecuredForm unit

This is the unit, which is intended to simplify the use of library. With it, you do not have to worry about creating TSecRights objects, initializing it or anything at all (although it is necessary to create a Security object with call to AddSecurity function). All you have to do is derive your form from this class (it is a good idea to put the SecuredForm into object repository, and then use inherit option to create new forms), and in the constructor of new form call constructor of SecuredForm with second parameter denoting the SecurityLib object (by name). The right that has the same name like the new form will automatically read during construction.

Then you can use several methods to set appropriate properties of the controls corresponding to the read rights. You can assign a function to the OnRightsRead event that will be called after rights have been read successfully. Also you can override virtual function RightsRead, which originally fires OnRightsRead event. Still, you may use Rights object in any place in your code – so you might just set the properties in the constructor of the new class.

After construction you may use two additional methods: Bind – that rebinds the Rights object to another Security object and ReadRights, which rereads rights from database and calls RightsRead.

DB Providers

What is it?

DB Providers are dll libraries that take care of connecting to the security database and retrieving data. One DB Provider may consist from many files and it can support many kinds of database connectivity. There is a list of DB Providers maintained in system registry. There may be many versions of one provider, each for different version of VCL. That version must match that of main SecurityLib file.

Registry

Names and paths of DB Providers are stored in registry in HKEY_LOCAL_MACHINE\SOFTWARE\MarVic\SecurityLib\DBProviders key. Each value name in this key corresponds to DB Provider name, and the value is interpreted as path to the DB Provider file. You may also omit path, then standard windows search procedure for dlls will be used.

It is recommended that names of DB Providers should be of following form: Vendor:DBSystem:VclX, where Vendor is short name of the company that produced the DB providers, DBSystem should denote the type of databases that the provider will use, and X corresponds to version of VCL.

Currently four names are used: MarVic:ADO:Vcl5, MarVic:ADO:Vcl4, MarVic:BDE:Vcl5, MarVic:BDE:Vcl4.

The setup program of your application should write the name and path of db provider into registry. It also should check whether there are no newer versions of shipped db provider already existing in the system.

Enclosed DB providers

Currently two DB Providers are available. The first one is for Borland Database Engine, which uses standard C++Builder components, and the second uses ADO, and is based on ADO Express components. There are two versions of BDE provider: for VCL 4 and for VCL 5. BDE Provider in VCL 4 version needs no additional libraries (all code is in vcldb40.bpl), and version for VCL 5 needs vclbde50.bpl library. ADO provider uses vclado50.bpl. You must ship these files with your application.

Documents

There are additional documents describing DB providers:

[“Custom database access”](#)

How to create new DB provider

[“BDE Provider”](#)

Description of BDE DB provider

[“ADO Provider”](#)

Description of ADO DB provider

Help files

The default location for help files is in Windows Help directory in subdir SecurityLib. But, this can be overridden by registry settings. The new path should be placed in key HKEY_LOCAL_MACHINE\SOFTWARE\MarVic\SecurityLib\1.0\HelpFilesLocation of type string.

Other

Other documents:

[“SecurityLib info”](#)

Basic information and introduction.

[“User’s Manual”](#)

Information for users of programs based on SecurityLib.

[“SecurityLib Administrator Manual”](#)

Information for administrators.

[“Interface and programming guide”](#)

Information for programmers.

www.marvic.prv.pl/securitylib/

SecurityLib website.

MS Access, Windows are trademarks of [Microsoft](#) Corp.

All other trademarks are owned by their respective owners and were used only in information purposes.